Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Advanced Tutorial One

Introduction to the Windows API and Icon management

Contents

Introduction **3**
Tutorial Requirements **3**
Part One: Introducing the Windows API **4**

Introduction

In our series of basic tutorials, we embarked on a journey to create a basic content launcher in C#. We finally reached the end of that journey in tutorial five and had our working (but bland) launcher to show for all the hard work we put in.

Now it's time to get serious. Let's look at how we can implement icons into our launcher, and in doing so, we'll visit the world of Windows API programming.

Tutorial Requirements

The advanced tutorials are enhancements to the first five basic tutorials that built our Universal Content Launcher. You can either work through these tutorials first, or download the source code, and have a go at this advanced tutorial.

For the purposes of this tutorial, it is assumed that you fully understand everything that is in the downloadable source code.

Part One: Introducing the Windows API

So, let's add some icons to our treeview control, to do this we need to add an ImageList control to our UCLMain form, and then tell UCL that this is the ImageList that we want the Treeview control to use. This is exactly the same as what we did when we added a context menu to UCL.

So:

- Go to the UCLMain design view.
- Drag the ImageList control from the toolbox onto your form design.
- Click on the Treeview control and scroll down the properties until you see ImageList, then select you ImageList control (ImageList1)

Great, so now we have told UCL to use this image list with outr treeview. We can also set some interesting things in the image list control, like Icon sizes and Bit Depths, set this to what you want by altering the properties of the ImageList control. If you're not sure what they should be, then select an image size of 16x16 and a bit depth of 32. You can play about with these settings and see how they change the look and feel of UCL.

Now we have the ImageList control on our form, we need to populate it with the icons that are stored in the content executables, and then attach the correct icons to the correct treeview nodes. This in practice is relatively simple, but to do it we need to write some new stuff to interface with the Windows API, because we have no other way of reading an icon from an executable file.

So, I want you to create a new class, and call it Utility.cs, as we will put any external methods we use, in here from now on.

Here is my complete Utility class, don't be scared, we're going to talk through it:

```
using System;
using System.Runtime.InteropServices;
using System.Drawing;

namespace UCL
{
    class Utility
    {

        [DllImport("shell32.dll")]
        static extern uint SHGetFileInfo(string fPath, uint fAttributes,
ref SHFILEINFO fi, uint fLength, uint fFlags);

        [StructLayout(LayoutKind.Sequential)]
        struct SHFILEINFO
        {
            public IntPtr hIcon;
            public int iIcon;
            public uint dwAttributes;
            public string szDisplayName;
            public string szTypeName;
        }

        static public Icon GetAppIcon(string appPath)
        {
            SHFILEINFO sfi = new SHFILEINFO();
            sfi.hIcon = IntPtr.Zero;
            if (SHGetFileInfo(appPath, 0, ref sfi,
(uint)Marshal.SizeOf(sfi), 256) != 0 && sfi.hIcon != IntPtr.Zero)
            {
                return Icon.FromHandle(sfi.hIcon);
            }
            else
            {
                return null;
            }
        }

    }
}
```

Don't panic!

For starters, we can jump right into the class, as we know what the `using`, `namespace` and `class` statements do. The next line is the statement that allows us to access a function (method) from an external file. This is what the Windows API is all about, reusing libraries of functions that have been written by someone else, so that we don't need to write things that are commonly used, just the specifics of our own program. In this case, we are re-using a windows function to allow us to read the icon that is embedded in an executable file. Let's look at the statement:

```
        [DllImport("shell32.dll")]
        static extern uint SHGetFileInfo(string fPath, uint fAttributes,
ref SHFILEINFO fi, uint fLength, uint fFlags);
```

The first line of this defines the actual dll file the function resides in, as it is a windows dll, and therefore its location is within the normal windows search path, then we do not need to define its exact location on our machine. If it was a 3rd party dll, then we would need to either copy it into the same directory that our executable resides in, or define the full path to it.

So, here we are telling UCL to look in `"shell32.dll"`

The next two lines are like a standard method call in C#, and these include declarations like `static` and the type of return value (`uint`). There is also the `extern` statement in there, and that tells C# that this is an external method and is not located within our project.

The name `SHGetFileInfo` is the actual function or method that exists in `shell32.dll`, and after we have identified this, we can then define the parameters the function uses in exactly the same way that we would an internal method.

All of these parameters should make sense, but you might take note of the `SHFILEINFO` type object, and that is covered in the next piece of the code:

```
[StructLayout(LayoutKind.Sequential)]
struct SHFILEINFO
{
    public IntPtr hIcon;
    public int iIcon;
    public uint dwAttributes;
    public string szDisplayName;
    public string szTypeName;
}
```

This code defines the `SHFILEINFO` type object. It is a lot like a Struct that we can define ourselves, and you will often find that an external function will require an object of a specific type to be passed to it.

If you look back at the `DllImport` statement, you will notice that the `StructLayout` object is passed by `ref`. When you pass something in this way, then the actual object from your code is passed into the function you are calling, this means that if the function executes as expected, then your object will have been populated or modified by the external function.

In more advanced external function calls, you will find many problems that can only be circumvented by Marshalling the objects and types that you pass to a function. As Marshalling is a big, big thing, that addresses the bridging between managed and unmanaged code, we are not going to cover it in these tutorials, but will touch upon some of its simpler uses. We can more or less avoid it for the task of extracting icons from executables; though as you will see, not entirely.

If you do not pass an object by `ref`, then you are merely passing its value to the external function, and your object or variable will remain unchanged.

Back to the `SHFILEINFO` object, and you will no doubt see that we define it as a `StructLayout` that is sequentially laid out, so all the objects that make up the `SHFILEINFO` object, are in the order presented above.

The rest of the variables in this `StructLayout` are mainly of types we already know of, but you may want to note that we have both `int` and `uint` numeric types, the difference being that a `uint` integer is unsigned, so it can not contain positive and negative values, but because of this, can hold a larger number than an `int`.

The other thing to note is this:

```
public IntPtr hIcon;
```

The `IntPtr` object is a special type of object that will no doubt be familiar to people who have messed around with C style programming languages. It is basically a pointer or handle to an object, so it shows C# where, in memory, an object resides; allowing direct access to it.

Next we have the actual method:

```
        static public Icon GetAppIcon(string appPath)
        {
            SHFILEINFO sfi = new SHFILEINFO();
            sfi.hIcon = IntPtr.Zero;
            if (SHGetFileInfo(appPath, 0, ref sfi,
(uint)Marshal.SizeOf(sfi), 256) != 0 && sfi.hIcon != IntPtr.Zero)
            {
                return Icon.FromHandle(sfi.hIcon);
            }
            else
            {
                return null;
            }
        }
```

Let's work through it.

```
            SHFILEINFO sfi = new SHFILEINFO();
```

Ok, create a new `SHFILEINFO` object called `sfi`, that's easy enough!

```
            sfi.hIcon = IntPtr.Zero;
```

In our new object, make sure the icon pointer is set to `Zero`, this will allow us to detect a valid icon when the external function returns our `sfi` object.

```
            if (SHGetFileInfo(appPath, 0, ref sfi,
(uint)Marshal.SizeOf(sfi), 256) != 0 && sfi.hIcon != IntPtr.Zero)
```

Ok, here we are both making the call and testing the result. The call itself is quite simple if you look at what we are passing, apart from a couple of things:

- Note we are passing the `sfi` object by `ref`.
- The number 256 in this call, is the identifier to return the Standard Icon contained in the executable, you can also return Large and Small Icons, but we would need to do more in our code to manage these, so we are just grabbing the standard one for now.
- This part: `(uint)Marshal.SizeOf(sfi)` is a very simple example of Marshalling something. What this is actually doing is getting the exact size of the `sfi` object we are passing, and then converting it into a `uint` variable, which is what the `SHGetFileInfo` function expects for that parameter.

You will often find that external functions, particularly those written in languages like C++, like you to tell them how big the data is that you are passing to them. This is because these languages do not offer automated memory management, so everything needs to be carefully calculated and allocated or crashes would become a very frequent thing.

Once we have done the call, we test that the function did not error by checking that its return value is not equal to zero, and we also test the icon pointer in our `sfi` object to check it now

contains something.

If it doesn't then our method will return `null`, if it does, then we do this:

```
                return Icon.FromHandle(sfi.hIcon);
```

We use the `FromHandle` method of the `Icon` object to retrieve an icon by its handle, or pointer that is returned in the `sfi` object. Don't forget that our method is designed to return an Ico, so this is fine.

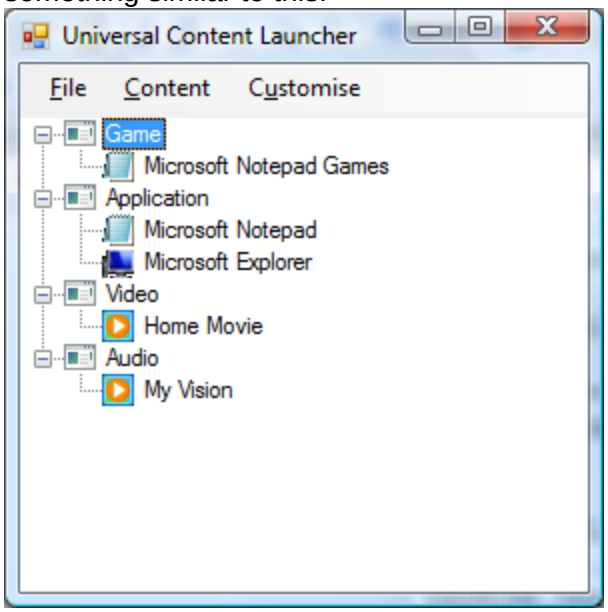
See, it's not as bad as it looks!

And now we've tackled that, we only need to put these icons into our ImageList and then assign the correct images from the ImageList to the relevant Treeview nodes, here's my complete `LoadContentList` method, with comments next to the parts I have changed:

```
        public void LoadContentList()
        {

            if (File.Exists(Path.Combine(Application.StartupPath,
"UCLContent.xml")))
            {
                XPathDocument doc = new
XPathDocument(Path.Combine(Application.StartupPath, "UCLContent.xml"));
                XPathNavigator nav = doc.CreateNavigator();
                int tIconCount = 0; //we need an icon count to keep track
of how many icons we have added to the ImageList
                Icon tIcon; //we need an Icon object to add the icons to
the ImageList

                imageList1.Images.Clear(); //make sure the ImageList is
clear
                treeView1.Nodes.Clear();

                TreeNode tNode;


                foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
                {
                    tNode = treeView1.Nodes.Add(tType);

                    XPathExpression expr;
                    expr = nav.Compile("/Content/" + tType + "/Name");
                    XPathNodeIterator iterator = nav.Select(expr);

                    try
                    {
                        while (iterator.MoveNext())
                        {

                            UCLContent tContent = new UCLContent();

                            XPathNavigator nav2 = iterator.Current.Clone();
                            Type type = tContent.GetType();
                            PropertyInfo tTemp =
type.GetProperty(nav2.Name);
                            tTemp.SetValue(tContent, nav2.Value, null);
```

```
                            TreeNode newNode =
tNode.Nodes.Add(tContent.Name); //before, we simply used this to add a new
node to the treeview, but now we need to store it in a treenode object
called newNode, so we can assign an image index to the node and therefore
link the correct image to the correct node

                            tTemp = type.GetProperty("Type");
                            UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
                            tTemp.SetValue(tContent, tUCLType, null);

                            while (nav2.MoveToNext())
                            {
                                tTemp = type.GetProperty(nav2.Name);
                                tTemp.SetValue(tContent, nav2.Value, null);
                            }

                            tIcon =
UCL.Utility.GetAppIcon(Path.Combine(tContent.ExePath , tContent.ExeName));
//this line calls our new method, and returns an icon for the executable
                            imageList1.Images.Add(tIcon); //now we add it
to our imagelist
                            newNode.ImageIndex = tIconCount; //and set the
node image list index to this new icon

                            newNode.SelectedImageIndex = tIconCount; //now
we set the nodes selected image list index to the new icon as well, or our
icon will change on screen to the default icon whenever we click on and
select a node

                            tIconCount++; //add 1 to the icon counter
                            tIcon = null; //as we are in a loop, release
the icon object and free the memory

                            UCLContentList.Add(tContent);

                            tContent = null;
                        }
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(ex.Message);
                        MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                    }

                }
            }
            else
            {
                MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
            }

            if (GetUCLSetting("Settings", "AutoExpand") == "True")
            {
                treeView1.ExpandAll();
            }
            SetTreeviewMode();

        }
```

So, hopefully you can follow all that and see exactly how it works. If we run UCL now, we get something similar to this:



So, points to note:

- We need some way of defining icons for the categories (Game / Application / Video / Audio)
- The Audio and Video categories display the icon of the media player, this may or may not seem correct depending on how you look at things.

So, let's define another way of grabbing and Icon, from an .ico file itself.

To do this we will create a folder in our UCL folder called Icons. We can then copy icons into this folder for use with our project, the idea being that if we find an .ico file in the Icons folder with a name that matches our category name, then we will use that icon. We could also extend this to work for content as well, so if an .ico file exists for some content, then that will take priority over the executables icon, and be displayed instead.

In the solution explorer, right click on UCL and select Add->New Folder and name it Icons.

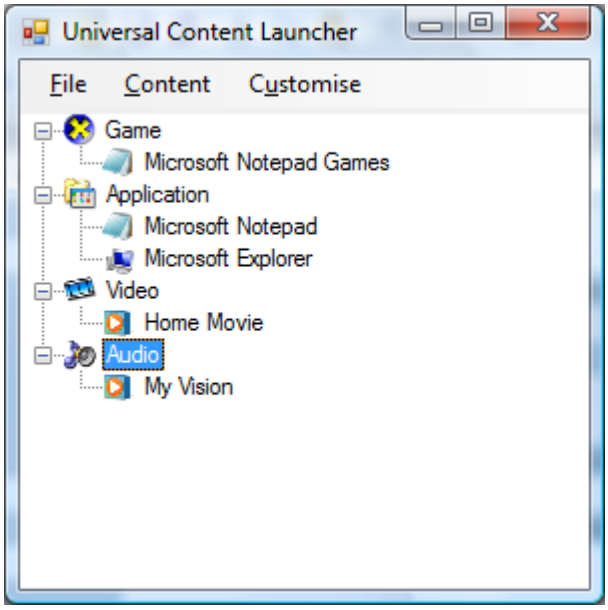
Now right click on the Icons folder, select Add and select some icons to add to the folder (I used the ones from the UGL Skin folder, but you can use any icons to test) but make sure you rename them so that they match the names of your categories. Also, make sure you tell UCL to copy them to the output folder, like we did with the XML files we created. You can change properties for multiple objects in the solution explorer by clicking on the first icon file, then holding down shift and clicking on the last icon file, then changing the property whilst they are all still highlighted.

To actually use the icon in our code is pretty similar to what we have already done, except that we need to first check if the file exists, and then create a new Icon based on this file in memory before setting our `tIcon` object to use it. The full code follows on the next page, look at it and you will see that this code will pull in an icon for either the category or the content, if it exists in the Icons folder. Look for the comments in the code below that show you what has changed.

```
        public void LoadContentList()
        {

            if (File.Exists(Path.Combine(Application.StartupPath,
"UCLContent.xml")))
            {
                XPathDocument doc = new
XPathDocument(Path.Combine(Application.StartupPath, "UCLContent.xml"));
                XPathNavigator nav = doc.CreateNavigator();
                int tIconCount = 0;
                Icon tIcon;

                imageList1.Images.Clear();
                treeView1.Nodes.Clear();
                TreeNode tNode;

                foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
                {
                    tNode = treeView1.Nodes.Add(tType);
                  //new code below
                    if (File.Exists(Path.Combine(Application.StartupPath +
"\\Icons\\" ,  tType + ".ico")))
                    {
                        tIcon = new
System.Drawing.Icon(Path.Combine(Application.StartupPath + "\\Icons\\",
tType + ".ico"));
                        imageList1.Images.Add(tIcon);
                        tNode.ImageIndex = tIconCount;
                        tNode.SelectedImageIndex = tIconCount;
                        tIconCount++;
                        tIcon = null;
                    }
                  //new code above

                    XPathExpression expr;
                    expr = nav.Compile("/Content/" + tType + "/Name");
                    XPathNodeIterator iterator = nav.Select(expr);

                    try
                    {
                        while (iterator.MoveNext())
                        {

                            UCLContent tContent = new UCLContent();

                            XPathNavigator nav2 = iterator.Current.Clone();
                            Type type = tContent.GetType();
                            PropertyInfo tTemp =
type.GetProperty(nav2.Name);
                            tTemp.SetValue(tContent, nav2.Value, null);
                            TreeNode newNode =
tNode.Nodes.Add(tContent.Name);

                            tTemp = type.GetProperty("Type");
                            UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
                            tTemp.SetValue(tContent, tUCLType, null);

                            while (nav2.MoveToNext())
                            {
```

```
                                tTemp = type.GetProperty(nav2.Name);
                                tTemp.SetValue(tContent, nav2.Value, null);
                            }
                        //new code below

                            if
(File.Exists(Path.Combine(Application.StartupPath + "\\Icons\\",
tContent.Name + ".ico")))
                            {
                                tIcon = new
System.Drawing.Icon(Path.Combine(Application.StartupPath + "\\Icons\\",
tContent.Name + ".ico"));
                                imageList1.Images.Add(tIcon);
                                tNode.ImageIndex = tIconCount;
                                tNode.SelectedImageIndex = tIconCount;
                                tIconCount++;
                                tIcon = null;
                            }
                            else
                            {

                                tIcon =
UCL.Utility.GetAppIcon(Path.Combine(tContent.ExePath, tContent.ExeName));
                                imageList1.Images.Add(tIcon);
                                newNode.ImageIndex = tIconCount;
                                newNode.SelectedImageIndex = tIconCount;
                                tIconCount++;
                                tIcon = null;
                            }

                        //new code above

                            UCLContentList.Add(tContent);

                            tContent = null;
                        }
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(ex.Message);
                        MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                    }

                }
            }
            else
            {
                MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
            }

            if (GetUCLSetting("Settings", "AutoExpand") == "True")
            {
                treeView1.ExpandAll();
            }
            SetTreeviewMode();

        }
```

And now this is what I see when I run UCL:



That's it for Icons, and this was a really quick advanced tutorial!

Hang on to your hats though, as next up we're going to be interfacing with a Steam Emulator to allow us to play all our Steam content offline and across a LAN directly from UCL! This will be in the form of a series of advanced tutorials that build on the concept of interfacing with external functions, and will reveal some insight into exactly how Steam works.

For now though, sit back, relax, and change some things in UCL yourself. Why not add user settings for switching icons on or off, or even a setting for changing Audio and Video content icons to match their category icons. You could even give your UCL executable its own Icon, and display this in the UCL forms title bar, why not have a look; it's really easy to do.

Don't be afraid to experiment, back up your code, try things out, and learn some more, there's lots that you can do with this base code, and doing things for yourself is always an effective way of learning.